Composition d'Informatique C (XULSR)

Filière MPI

1 Méthode de correction et barème

Chaque question rapporte un certain nombre de *points* et est notée avec un *score* entre 0 et 5. Le nombre de points total est de 49, et un coefficient multiplicateur de 1,1 est utilisé pour obtenir la note finale, en tronquant à 20. En d'autres termes, 44.6 points étaient suffisants pour obtenir 20/20, et la note finale sur 20 d'un·e candidat·e est alors obtenue en appliquant la formule

$$\text{note(copie)} = \max\left(20,\ 1.1 \times \frac{20}{49} \times \sum_{\text{question } q} \text{points}(q) \times \frac{\text{score(copie}, q)}{5}\right).$$

La moyenne des 396 copies est de 9,38/20 avec un écart type de 3,86.

2 Commentaires généraux et conseils aux candidat·es

Il semble nécessaire de commencer par rappeler qu'il est fondamental de lire le préambule de chaque sujet, et que ce n'est pas du temps perdu : il y avait cette année une liste de fonctions autorisées en OCaml ; les autres étaient interdites, dont `List.iter`. Nous avons retiré des points en cas d'utilisation de fonctions non autorisées.

Nous soulignons ensuite l'importance de rendre une copie propre et lisible. La correctrice ou le correcteur ne doit pas avoir à faire d'effort pour lire votre copie. Nous insistons sur le fait qu'en cas de doute sur ce qui est écrit, les points ne seront pas distribués. La présentation de certaines copies que nous avons corrigées était inacceptable pour un concours d'entrée aux grandes écoles.

À l'inverse, les questions théoriques demandant plus d'une page de rédaction sont extrêmement rares, et celles demandant plus d'une demi-page de justification sont rares. Soyez attentives et attentifs à ne pas vous égarer dans la rédaction et réfléchissez avant d'écrire. Les démonstrations par l'absurde ne sont pas forcément les plus simples : en particulier, il est toujours possible de se passer d'un raisonnement par l'absurde. Préférez si possible les démonstrations constructives, plus faciles à rédiger.

Concernant les questions de code, il est fondamental de choisir des noms simples et éclairants pour les noms de fonctions et de variables. Pour donner un exemple, un paramètre de type `foret` ne peut raisonnablement pas s'appeler `arbre`. Nous conseillons également d'éviter les pattern-matching sur une liste de la forme `a::b::c`, qui laissent penser dans la suite que `c` est un élément de la liste au même titre que `a` ou `b`.

De façon générale, il est primordial de s'assurer que la correctrice ou le correcteur comprenne ce que vous faites, pour qu'elle ou il puisse vous évaluer à votre juste valeur. Ainsi un code de plus d'une page ne peut pas se passer de commentaires. Ne vous attendez pas à ce que le ou la correctrice comprenne ce que fait une fonction auxiliaire d'après son simple nom, et encore moins lorsque la fonction est anonyme. Ainsi l'abus de l'utilisation de `List.fold_left` (qui était par ailleurs interdite dans le préambule) avec des fonctions auxiliaires anonymes et sans aucune explication ne peut que nuire à la juste évaluation de votre réponse.

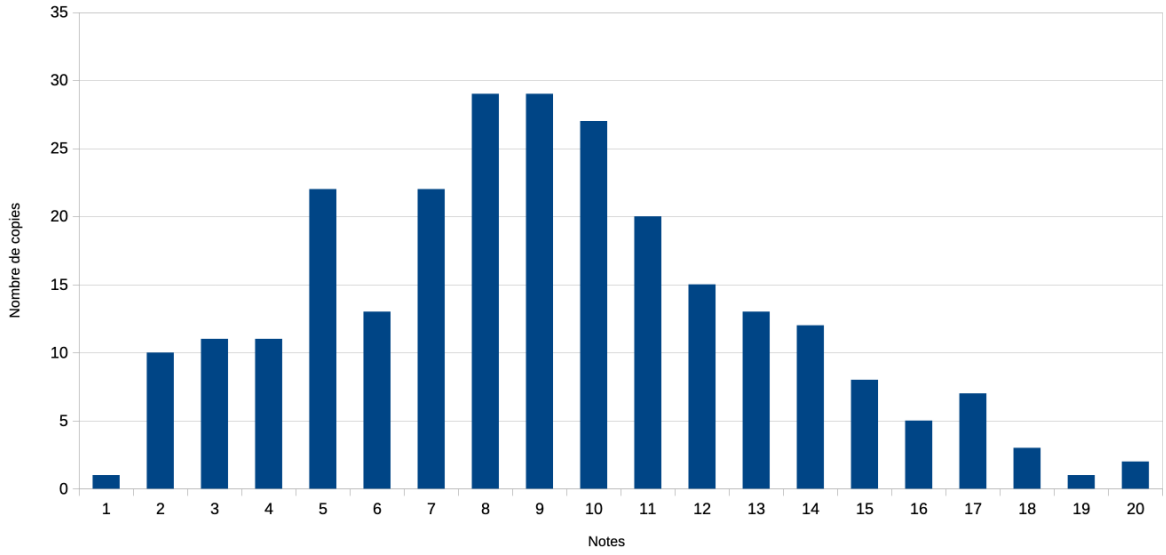

FIGURE 1 – Histogramme des notes

Les premières questions d'une partie sont généralement des questions de compréhension pour vous aider à vous approprier les notions introduites dans le sujet. Il ne faut pas supposer que ce sont des questions pièges, ou qu'il y a une erreur dans le sujet, c'est très improbable au début de l'énoncé. Ce sont souvent des questions auxquelles il faut répondre proprement et rapidement.

Enfin nous ne pouvons que recommander aux candidat·es de vérifier leur code sur des exemples simples, des listes à un ou deux éléments, afin de vérifier notamment les conditions d'arrêt des fonctions récursives.

3 Commentaire détaillé

Pour chaque question, sont indiqués entre crochets : le pourcentage de copies ayant répondu à la question, le nombre de points de la question, et la moyenne des scores (entre 0 et 5) des copies ayant traité la question. Les pénalités indiquées, qui portent sur le score, sont données de façon indicative à titre d'exemple et reflètent les erreurs les plus communes.

3.1 Partie I

Question 1 [100% • 0,5 • 4,62]. Question facile bien traitée. Les copies utilisant `List.filter` ont été pénalisées (-2). Certaines copies ont renvoyé le complémentaire de la liste demandée, ou se sont arrêtées dès le premier élément commun trouvé, et ont perdu tous les points à la question.

Question 2 [100% • 0,5 • 4,85]. Cette question facile a été également bien traitée, même si certaines copies ont renvoyé les valeurs au lieu des clefs (-2,5).

Question 3 [100% • 0,5 • 4,5]. Une justification rapide de la réponse était attendue.

Question 4 [100% • 0,5 • 4,71]. Il était supposé que `cpl` était un couplage, donc par définition sans intersection, il était ainsi inutile de faire appel à `join` pour supprimer les doublons, au risque d'avoir une mauvaise complexité (-1).

Question 5 [100% • 0,5 • 4,96]. Rien à signaler.

Question 6 [98% • 0,5 • 4,45]. Certaines copies ont à tort pensé que le couplage proposé était maximal et n'avait donc pas de chemin d'augmentation. Comme nous l'avons dit dans les commentaires généraux, il est très peu probable qu'il y ait une erreur sur une question de compréhension du sujet.

Question 7 [99% • 0,5 • 4,88]. Question très bien traitée. Il ne fallait pas oublier le chemin d'augmentation $1 \to 9$.

Question 8 [97% • 1,5 • 3,68]. 3,5 points portaient sur la démonstration que $C(P)$ était bien un couplage, et 1,5 point sur le fait qu'il contenait plus d'arêtes que C. Plusieurs copies ont oublié la deuxième partie de la question, perdant des points inutilement. Certaines copies supposaient que le chemin d'augmentation contenait forcément une arête du couplage C, ce qui est faux dans le cas où le chemin d'augmentation contient une seule arête (comme $1 \to 9$ dans la question 7). Plusieurs copies ont utilisé un raisonnement par l'absurde pour montrer que $C(P\ )$ est un couplage, compliquant inutilement la rédaction, et se sont perdues dans des pages de cas particuliers mal organisés. Il suffisait pourtant simplement de vérifier proprement les hypothèses de la définition d'un couplage.

Question 9 [99% • 1,5 • 3,56]. Il était précisé dans le sujet qu'une arête $\{a, b\}$ était représentée par une paire ordonnée `(min a b, max a b)`, il fallait donc penser à ordonner les arêtes (-1). Plusieurs copies ont traité un chemin comme une liste d'arêtes au lieu d'une liste de sommets, leur faisant perdre 2 points. Nous avons également pénalisé (-1) les copies qui ont interverti `lstin` et `lstout`, sauf si leur utilisation était cohérente à la question 10. En revanche, les copies pour lesquelles `lstin` et `lstout` étaient inversées de manière incohérente, notamment selon la parité de la longueur du chemin, ont perdu des points (-1). Il y a eu également des erreurs sur les appels récursifs, qui ne prenaient pas toujours en compte qu'un sommet intérieur au chemin appartenait à deux arêtes, si bien que certaines arêtes étaient perdues (-2). Nous insistons sur le fait qu'il suffisait de tester rapidement son code sur de petits exemples simples pour identifier et réparer ces problèmes.

Question 10 [98% • 1 • 3,7]. Pas de difficulté particulière, si on avait bien compris les rôles de `lstin` et `lstout` (qu'il ne fallait pas inverser (-2)), et les différentes fonctions à disposition (comme `del` et `join`) pour manipuler des ensembles.

Question 11 [94% • 4 • 2,93]. Il s'agissait d'une question plus difficile, qui demandait trois choses : rassembler les voisins de `w` pour créer sa liste d'adjacence (3 points), renommer les sommets de `lst` en `w` dans les listes d'adjacence des autres sommets (1 point), et enfin supprimer les sommets de `lst` des sommets du graphe contracté (1 point) ; tout cela en évitant les doublons (-1). Un exemple de correction :

```
let contracteG graphe lst w =
  let rec parcoursG voisinsw partgraphe = match partgraphe with
    | (sommet,adj) :: q when List.mem sommet lst
                        -> let new_voisinsw = join voisinsw (del adj lst)
                           in parcoursG new_voisinsw q
    | (sommet,adj) :: q -> (sommet,renomme adj lst w)::(parcoursG voisinsw q)
    | [] -> [(w, voisinsw)]
```

```
in
parcoursG [] graphe;;
```

Question 12 [88% • 1 • 4,08]. Il était essentiel que le raisonnement explique dans la preuve que le couplage C contenait au plus une arête joignant un sommet du bourgeon à un sommet extérieur au bourgeon (-3). Là encore un raisonnement par l'absurde n'était pas idéal pour attaquer la question, et plusieurs copies se sont perdues dans des énumérations de cas qui leur ont fait perdre beaucoup de temps.

Question 13 [88% • 1 • 3,69]. Il fallait ici aussi penser à ordonner les arêtes avec `(min a b, max a b)` (-1). La fonction demandée devait parcourir le couplage en traitant plusieurs cas (-1 par cas mal traité) : éliminer les arêtes qui concernent deux sommets du bourgeon, renommer un sommet du bourgeon par `w` si l'arête correspondante contient un seul sommet du bourgeon, garder inchangées les autres arêtes du couplage.

Question 14 [85% • 2 • 3,58]. Un certain nombre de copies ne comprend pas comment fonctionnent les types récursifs en OCaml. Nous avons pénalisé (-1) l'oubli du `N` pour décrire un arbre. De même, certaines copies ne maîtrisent pas non plus le type `Option` et la syntaxe `None/Some` (-1). L'ajout de mauvais sommets dans le chemin a été pénalisé (-2), ainsi que l'oubli du sommet `v` (-1). Il fallait également penser selon la façon dont le code était écrit à inverser la liste si le chemin était construit à l'envers (-1). Certaines copies ont perdu des points (-2) pour de mauvais appels récursifs ; il fallait notamment continuer à chercher dans les autres arbres de la forêt si l'élément `v` n'avait pas été trouvé dans le premier arbre. Il était judicieux (mais pas obligatoire) pour cette question de créer une fonction auxiliaire `find_arbre`.

Question 15 [80% • 2 • 3,86]. Les copies qui avaient réussi la question 14 ont en général réussi cette question également. Nous avons observé certaines erreurs récurrentes, comme `v` qui était ajouté à la forêt au lieu de `N(v,[])` (-1), la structure d'arbre ou de forêt qui était détruite au cours des appels récursifs (-2), ou encore des appels récursifs oubliés lorsque la racine considérée n'est pas `u` (-2).

Question 16 [50% • 4 • 2,45]. Cette question était plus délicate, et demandait d'identifier deux propriétés pour avoir tous les points : il fallait remarquer et justifier proprement dans un premier temps que pour tout arbre A dans la forêt F, et tout noeud b dans A, le chemin de b à la racine de A est un chemin alternant entre des arêtes du couplage C et des arêtes hors du couplage, dont la dernière arête contenant la racine n'est pas dans le couplage (2,5 points). Il fallait également justifier qu'un sommet de G apparaît au plus une fois dans F (2,5 points) pour vérifier que le chemin proposé est bien élémentaire, ce qui a souvent été oublié.

Question 17 [62% • 1 • 3,69]. Cette question demandait de bien comprendre le lien entre l'algorithme proposé à la question 16, et son implémentation partielle en Ocaml proposée à la figure 1. Pour cette question, il fallait remarquer que la ligne 15 correspondait au cas (b), où le voisin `v` de `u` se trouve dans la forêt F, mais est à profondeur impaire, donc aucune des conditions i) ou ii) n'était vérifiée. Il fallait donc ne rien faire et continuer de parcourir les voisins suivants de u : `aux_voisins u tl`.

Question 18 [61% • 1 • 4,12]. On se trouve dans le cas (3.b.ii), il fallait renvoyer le chemin d'augmentation correspondant :

```
let aug = cu@(List.rev cv) in
Some aug
```

Les oublis de `Some` ont été pénalisés (-1), ainsi que l'oubli d'inversion de cv (-2).

Question 19 [52% • 4 • 1,77]. Il aurait été judicieux pour beaucoup de copies de faire un dessin pour visualiser ce que la question demandait de faire. Plusieurs copies ont perdu beaucoup de points (-4) en considérant que les deux chemins `cu` et `cv` n'avaient que la racine comme sommet en commun, et excluaient donc que la racine de l'arbre pouvait ne pas se trouver dans le bourgeon.

Question 20 [51% • 2 • 2,94]. Cette question fait référence au cas 3.*b.ii*). Il fallait donc chercher récursivement un chemin augmentant dans le graphe contracté `nc`, pour ensuite le convertir en un chemin d'augmentation (avec `gonfle`) pour le graphe de départ. La fonction `recherche` renvoie une option, il y avait donc deux cas à traiter, selon qu'elle renvoie `None` (2 points) ou `Some(ch)` (3 points). Il fallait également penser à renvoyer une option (-2).

Question 21 [56% • 2 • 4,07]. La plupart des copies qui ont traité cette question ont compris qu'il fallait partir d'un couplage vide, et augmenter successivement le couplage en cherchant un chemin d'augmentation, à l'aide de `recherche` et `augmente`. Nous avons pénalisé à nouveau les copies qui ont oublié que `recherche` renvoyait une option (-2).

3.2 Partie II

Question 22 [94% • 0,5 • 4,9]. Question très simple de compréhension, très bien traitée. Nous avons pénalisé une copie qui retournait A à la fin de `write_sqmatrix` (-1).

Question 23 [96% • 0,5 • 4,92]. Question bien réussie, avec à de rares occasions une inversion dans le sens des flèches.

Question 24 [95% • 0,5 • 4,84]. Question bien réussie.

Question 25 [94% • 0,5 • 3,61]. 3 points pour les marches fermées, 2 points pour les suites de marches fermées. Nous avons retiré 1 point par marche fermée manquante, -0,75 par suite de marche manquante. Plusieurs copies ont mal calculé les poids, en les additionnant au lieu de les multiplier (-2). Nous avons pénalisé (-1) les copies qui proposaient des marches ou suites qui n'étaient pas fermées.

Question 26 [80% • 0,5 • 3,8]. -1 par arc manquant, -1 par arc en trop, -1 par poids incorrect.

Question 27 [59% • 1 • 2,96]. Nous avons noté la première partie de la question (montrer qu'il s'agit d'un chemin dans H_A) sur 2 points, et la deuxième partie de la question sur 3 points, dont 1,5 point sur le traitement des jonctions au sein de H_A entre les différentes marches de la suite de marches fermées positive, et 1,5 points sur la jonction entre la fin de la dernière marche de la suite et t_0. Il était important de justifier proprement l'existence des arêtes dans H_A dont on avait besoin pour construire le chemin de s à t_0, en précisant notamment à quels moments intervenaient les hypothèses que la marche était fermée, et que la suite de marche était positive.

Question 28 [48% • 2 • 2,14]. Il fallait distinguer deux cas, selon que $v = h$ ou non :

— Si $v \neq h$, $F_A(\langle p, h, v, i) = \sum_{w \geq h}^{n-1} A_{w,v} F_A(\langle p, h, w, i-1)$.

— Sinon, $F_A(\langle p, h, h, i) = \sum_{h' < h, w > h'}^{n-1} A_{w,h'} F_A(\langle 1-p, h', w, i-1)$.

Nous avons notamment pénalisé les formules pour lesquelles les poids de la matrice étaient additionnés au lieu d'être multipliés (-2), les formules qui ne faisaient pas intervenir les poids de la matrice (-3), ou les erreurs d'indices dans les sommes (entre -1 et -2). Certaines copies ont tenté d'écrire une seule formule, avec des fonctions indicatrices, mais se sont généralement trompées dans leur utilisation (-3).

Question 29 [46% • 1 • 3,91]. Il s'agit d'une question simple, plutôt bien traitée. Il suffisait de créer un `Fourtable` avec `create` et d'associer 1 aux sommets de la forme $\langle n \mod 2, h, h, 0\rangle$. Nous avons pénalisé les copies qui ont omis de calculer $n \mod 2$ (-2), qui ont mis à 1 trop d'éléments de la couche 0 (par exemple en faisant une double boucle sur (h, u) au lieu d'une simple boucle sur (h, h)) (-4), ou se sont trompées dans l'ordre des paramètres lors de l'appel à `write` (-1).

Question 30 [31% • 4 • 2,78]. Nous n'avons pas pénalisé deux fois une même erreur : si le code calculait bien la formule proposée à la question 28, avec la bonne complexité, alors les points ont été accordés. Il y avait 3 points pour le code, et 2 points pour la justification de complexité. Il fallait être vigilent·e sur l'ordre des boucles imbriquées, certaines combinaisons n'étaient pas valides (-2). Certaines copies ont raté l'implémentation d'une somme, en écrasant la valeur précédente par la nouvelle au lieu de l'ajouter (-1), ou ont commencé leur boucle à la couche 0 qui avait été pourtant initialisée à la question précédente (-1). La justification de complexité était simple, mais a parfois été oubliée.

Question 31 [22% • 1,5 • 3,53]. 3 points pour le code, 2 points pour la complexité. Il fallait suivre la stratégie proposée après la question 27 pour implémenter la formule du déterminant page 23. La fonction `remplit` de la question précédente compte les chemins (pondérés) de s à n'importe quel sommet x dans H_A, il restait donc à simuler les arcs joignant ces sommets à t_0 et t_1 pour calculer le déterminant :

```
int determinant (int n, int * A)
{
  int pos = 0;
  int neg = 0;
  int h, v;
  Fourtable * t = remplit(n,A);
  for (h=0;h<n;h++){
    for (v=h;v<n;v++){
      pos += read(t,1,h,v,n-1)*read_sqmatrix(n,A,v,h);
      neg += read(t,0,h,v,n-1)*read_sqmatrix(n,A,v,h);
    }
  }
  free(t);
  return pos - neg;
}
```

Parmi les erreurs courantes, nous avons pénalisé l'oubli des poids de la matrice (-2), l'oubli de l'appel à `free` pour libérer la mémoire allouée dans le tas (-0,5), l'oubli du signe des suites de marches (-2).

3.3 Partie III

Question 32 [22% • 1,5 • 2,98] Cette question n'était pas très difficile, mais plusieurs copies ont fait deux tirages pour $i < j$ et $j < i$ alors que la matrice $T(G)$ doit être antisymétrique (-1). Plus généralement, nous avons pénalisé chaque propriété mal implémentée (-1) d'une matrice de Tutte. Nous avons pénalisé les copies qui modifiaient la matrice A (-3) ou oubliaient de calculer et renvoyer le déterminant (-1).

Question 33 [12% • 4 • 2,92] 2 points pour le code, 2 points pour justifier l'erreur, 1 point pour la complexité. Les raisonnements probabilistes n'étaient pas toujours rigoureux, avec quelques fois des confusions sur d'où venait l'aléa. Certains points ont été perdus inutilement sur le code, sans doute faute de temps, avec des conditions manquantes pour l'arrêt des boucles (-2), ou des fonctions qui renvoient la négation du résultat attendu (-1).